09/719399

REC'D 20 JUL 1999
WIPO PCT

PCT/NZ99/00081

S

Intellectual Property Office
of New Zealand
Te Pou Rāhui Hanga Hou

CERTIFICATE

This certificate is issued in support of an application for Patent registration in a country outside New Zealand pursuant to the Patents Act 1953 and the Regulations thereunder.

I hereby certify that the annexed is a true copy of the Provisional Specification as filed on 10 June 1998 with an application for Letters Patent number 330675 made by Auckland Uniservices Ltd.

Dated 05 July 1999.

Neville Harris

Neville Harris
Commissioner of Patents

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN
COMPLIANCE WITH
RULE 17.1(a) OR (b)

330675

Patents Form No. 4 Our Ref: AS801714

Patents Act 1953

PROVISIONAL SPECIFICATION

SOFTWARE WATERMARKING TECHNIQUES

We, **AUCKLAND UNISERVICES LIMITED**, a New Zealand company, of 58 Symonds Street, Auckland, New Zealand do hereby declare this invention to be described in the following statement:

INTELLECTUAL PROPERTY OFFICE
OF N.Z.
10 JUN 1998
RECEIVED

PT0469321

SOFTWARE WATERMARKING TECHNIQUES

FIELD OF THE INVENTION

The present invention relates to methods for protecting software against theft, establishing/proving ownership of software and validating software. More particularly, although not exclusively, the present invention provides for methods for watermarking what will be generically referred to as software objects. In this context, software objects may be understood to include programs and certain types of media.

BACKGROUND TO THE INVENTION

Watermarking is the process of embedding a secret message, the *watermark*, into a cover or overt message. For example, in media watermarking, the secret is commonly a copyright notice and the cover is a digital image, video or audio recording.

The watermark is constructed to make it difficult to detect and/or remove it without access to a secret key.

Watermarking a software object (hereafter referred to as an *object*) discourages intellectual property theft. A further application is that watermarking an object can be used to establish and/or prove evidence of ownership of an object. Fingerprinting is similar to watermarking except a different watermark is embedded in every cover message thus providing a unique fingerprint for every object. Watermarking is therefore a subset of fingerprinting and the latter may be used to detect not the fact that a theft has occurred, but may also allow identification of the particular object and thus establish an audit trail which can be used to reveal the infringer of copyright.

In the context of prior art watermark techniques, the following scenario serves to illustrate the ways in which a watermarked object may be vulnerable to attack. Suppose that A watermarks an object O with a watermark W and key K. If the object O *is* sold to B and B wishes to (illegally) on-sell O to C, there are various types of attack to which O may be vulnerable.

Detection: initially B must try and detect the presence of the watermark in *O*. If there is no watermark, no further action is necessary.

Locate and remove: once B has determined that *O* carries a watermark, C may try to locate and remove *W* without otherwise harming the rest of the contents of O.

Distort: if some degradation in quality of O is acceptable, B may distort it sufficiently so at it becomes impossible for A to detect the presence of the watermark *W* in the object *O*.

Add: alternatively, if removing the watermark *W* is too difficult, or distorting the object *O* is not acceptable, B might simply add his own watermark *W'* (or several such marks) to the object *O*. This way, A's mark becomes just one of many.

It is considered that most media watermarking schemes are vulnerable to attack by distortion. For example, image transforms such as cropping and lossy compression will distort the image sufficiently to render many watermarks unrecoverable.

To the knowledge of the applicants there exists no effective watermarking scheme which is capable of use with or appropriate for software. It would be a significant advantage to be able to apply watermarking techniques to software in view of the widespread occurrence of software piracy. It is estimated at software piracy costs approximately 15 billion dollars per year. Thus the problem of software security and protection is of significant commercial importance.

One simple way, known in the prior art, of embedding a watermark in a piece of software is simply to include it in the initialized static data section of the object code. In a similar, yet more complex manner, watermarks are often encoded in what is known as an " Easter egg". This is a piece of code, which is activated for a highly unusual or seldom encountered input to the particular application.

Thus, it is an object of the present invention to provide methods for watermarking software objects which overcomes the limitations inherent in prior art watermarking techniques and allows for non-media objects to be watermarked effectively. It is a further object of the present invention to provide methods for watermarking software objects which are resistant to the aforementioned techniques for attacking watermark objects or to at least provide the public with a useful choice.

DISCLOSURE OF THE INVENTION

In one aspect, the invention provides for a method of watermarking a software object whereby a watermark is stored in the state of the software object as it is being run with a particular input sequence.

The software object may be a program or piece of program.

The state of the software object may correspond to be stack, heap, global variables and the like, of the object.

In a preferred embodiment, the watermark may be stored in an objects' execution state whereby an input sequence *I* is constructed which, when fed to an application of which the object is a part, will make the object *O* enter a state which represents the watermark the representation being validated or checked by examining the global variables, heap, and/ or stack of the object *O*.

In an alternative embodiment, the watermark could be embedded in the execution trace of the object *O* whereby, as a special input *I* is fed to *O*, the address/operator trace is monitored and, based on a property of the trace, a watermark is extracted.

In a preferred embodiment, the watermark is embedded in the state of the program as it is being run with a particular input sequence $I=I_1,\dots I_k$.

The watermark may be embedded in the topology of a dynamically built graph structure.

The graph structure (or watermark graph) corresponds to a representation of the logical structure of the program and may be viewed as a set of nodes together with a set of vertices.

The method may further comprise building a recognizer *R* concurrently with the input *I* and watermark *W*.

Preferably *R* is a function adapted to identify and extract the watermark graph from all other dynamic structures on the heap or stack.

Preferably the watermark *W* may incorporate a marker that will allow *R* to recognize it easily.

In a preferred embodiment, *R* is retained separately from the program whereby *R* is dynamically linked with the program when it is checked for the existence of a watermark.

Preferably the application of which the object forms a part is obfuscated or incorporates tamper-proofing code.

n is a signature property of the watermark.

In a preferred embodiment, the method includes the creation of a number *n* wherein n is the product of two prime numbers, whereby *R* checks W for *n* which has been embedded in the topology of *W*.

The invention further provides for a method of verifying the integrity or origin of a program comprising:
embedding a watermark *W* in the state of a program as the program is being run with a particular input sequence I_k;
building a recognizer *R* concurrently with the input I_k and watermark *W* wherein the recognizer is adapted to extract the watermark graph from other dynamic structures on the heap or stack wherein *R* is kept separately from the program; wherein *R* is adapted to check for a number *n*, *n,* in a preferred embodiment, being the product of 2 primes and wherein *n* is embedded in the topology of *W*.

Other properties of *s(W) may be used to compute the signature.*

The number *n* may be derived from any combination of numbers depending on the context and application.

Preferably the program or code is further adapted to the resistant to tampering, preferably by means of obfuscation or by adding tamper-proofing code.

In a broad aspect, the recognizing *R* checks for the effect of the watermarking code on the execution state of the application thereby preserving the ability to recognizer the watermark in cases where semantics-preserving transformations have been applied to the application.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will now be described by way of example only and with reference to the figures in which:

Figure 1:	illustrates methods of adding a watermark to an object and attacking the integrity of such a watermark;
Figure 2:	illustrates methods of embedding a watermark in a program;
Figure 3:	illustrates an example of a function used to embed a watermark within a static string;
Figure 4:	illustrates insertion of a bogus predicate into a program;
Figure 5:	illustrates splitting variables;
Figure 6:	illustrates merging variables;
Figure 7:	illustrates the conversion of a code section into a different cirtual machine code;

Figure 8: illustrates an example of a method of the watermarking scheme according to the present invention;

Figure 9: illustrates a possible encoding method for embedding a number in the topology of a graph;

Figure 10: illustrates an another possible embodiment for embedding a number in the topology of a graph;

Figure 11; illustrates a marker in a graph;

Figure 12: illustrates examples of obfuscating transformation and

Figure 13: illustrates examples of tamperproofing Java code.

Referring to Figure 1(b) a way is shown by which Bob can circumvent a watermarking scheme by distorting the protected object. If the distortion is at ``just the right level", *O* will still be usable by Bob, but Charles will be unable to extract the watermark. In Figure 2(9), the distortion is so severe that *O* is not usable by Bob.

It has been shown that there are transformations that will effectively destroy most any kind of program structure. As a consequence, any software watermarking technique must be evaluated with respect to its resilience to attack from automatic obfuscation. The following discussion will survey obfuscating transformations that can be used to destroy software watermarks.

In Figure 2a a watermark is embedded within a static string. There are several ways of destroying such static data, the most effective perhaps is to convert it into a program that produces the data. As an example, consider the function G in Figure 3. This function was constructed to obfuscate the strings "AAA", "BAAAA", and "CCB". The values produced by G are G(1)="AAA", G(2)="BAAAA", G(3)=G(5)="CCB", and G(4)="XCB".

In Figure 2b Alice embeds a watermark within the program code itself. There are numerous ways to attack such code. Figure 4, for example, shows how it is possible to insert bogus predicates into a program. These predicates are called opaque since their outcome is known at obfuscation time, but difficult to deduce otherwise. Highly resilient opaque predicates can be constructed using hard static analysis problems such as aliasing.

In Figure 2 c a watermark is embedded within the state (global, heap, and stack data, etc.) of the program as it is being run with a particular input *I*. Different obfuscation techniques can be employed to destroy this state, depending on the type of the data. For example, one variable can be split into several variables (Figure 5) or several variables can be merged into one (Figure 6).

In Figure 2 d a watermark is embedded within the trace (either instructions or addresses, or both) of the program as it is being run with a particular input *I*. Many of the same transformations that can be used to obfuscate code will also obfuscate an instruction trace. Figure 7 shows another, more potent, transformation. The idea is to convert a section of code (Java bytecode in our case) into a different virtual machine code. The new code is then executed by a virtual machine interpreter included with the obfuscated application. The execution trace of the new virtual machine running the obfuscated program will be completely different from that of the original program. In Figure 2e, a watermark is embedded in an Easter Egg. Unless the code is obfuscated, Easter Eggs may be found by straightforeward techniques such as decompilation and dissassembly.

In this section, techniques for embedding software watermarks in dynamic data structures are discussed. The inventors view these techniques are the most promising for withstanding de-watermarking attacks by obfuscation.

The basic structure of the proposed watermarking technique is outlined in Figure 8. The method is as follows:

1. The watermark *W* is embedded, not in the static structure of the program (its code (Unix text segment), its static data (Unix initialized data segment), or its type information (Unix symbol segment or Java's Constant Pool), but rather in the state of the program as it is being run with a particular input sequence $I = I_1 \ldots I_k$.
2. More specifically, the watermark is embedded in the topology of a dynamically built graph structure. It is believed that obfuscating the topology of a graph is fundamentally more difficult than obfuscating other types of data. Moreover, it is anticipated that tamperproofing such a structure should be easier than tamperproofing code or static data. This is particularly true of languages like Java, where a program has no direct access to its own code.

3. A Recognizer *R* is built along with the input *I* and watermark *W*. *R* is a function that is able to identify and extract the watermark graph from among all other dynamic structures on the heap or stack. Since, in general, sub-graph isomorphism is a difficult problem, in a preferred embodiment it is possible that *W* will have some special marker that will allow to recognize *R* easily. Alternatively, *W* may be the entire dynamic state, i.e. markers may not be necessary.

4. An important aspect of the proposed technique is that *R* is not distributed along with the rest of the program. If it were, a potential adversary could identify and decompile it, and discover the relevant property of *W*. Rather, *R* is linked in dynamically with the program when we check for the watermark.

5. It is required that some signature property *s(W)* of *W* be highly resilient to tampering. This can be achieved, for example, by obfuscation or by adding tamperproofing code to the application.

6. In Figure 8 it is assumed that the signature *s(W)* that *R* checks for is a number *n*, which has been embedded in the topology of *W*. *n* is the product of two large primes *P* and *Q*. To prove the legal origin of the program, we link in *R*, run the resulting program with *I* as input, and show that we can factor the number that *R* produces. Alternatively, s(*W*) can be based on hard computational problems other than factorisation of large primes.

The above issues will now be discussed in more detail. The first problem to be solved is how to embed a number in the topology of a graph. There are a number of ways of doing this, and, in fact, a watermarking tool should have a library of many such techniques to choose from. Figure 9 illustrates one possible encoding. The structure is basically a linked list with an extra pointer field which encodes a base-6 digit. A null-pointer encodes a 0, a self-pointer a 0, a pointer to the next node encodes a 1, etc.

In the previous paragraph, it was shown how an integer *n* could be encoded in the topology of a graph. The encoding is resilient to tampering, as long as the recognizer *R* is able to correctly identify the nodes containing the two pointer fields in which we have encoded *n*. We now describe another encoding showing that a recognizer *R* can evaluate *n* = *s(W)* if it can identify only a single pointer field per node.

The node fields in *W* define a graph *G* such that all nodes in *G* have out-degree one. The strongly connected components of *G* can be found in *O(|G|)* time, using standard algorithmic techniques. Furthermore, in the same linear-time algorithm, our recognizer can discover the edges that link the strongly-connected components.

We shrink each strongly-connected component of *G* to a single node, constructing a reduced graph *G'*. We then compute the index *n* of *G'* in any convenient enumeration. For example, we might construct *W* so that *G* has at least one ``sink" node that is reachable by directed paths from all other nodes in *G*. In this case, *G'* is an oriented tree, so it is enumerable by the techniques described in Knuth, Vol I 3rd Edition, Section 2.3.4.4.

The number a_m of oriented trees with m nodes is asymptotically $a_m = c(1/\alpha)^{n-1}/n^{3/2} + O((1/\alpha)^n /n^{5/2})$ for $c \sim 0.44$ and $1/\alpha \sim 2.956$. Thus we can encode an arbitrary 1000-bit integer n in a graphic watermark W with $1000/\log_2 2.956 \sim 640$ nodes. nodes.

We construct an index n for any enumerable graph in the usual way, that is, by ordering the operations in the enumeration. For example, we might index the trees with m nodes in ``largest subtree first" order, in which case the path of length $m-1$ would be assigned index 1. Indices 2 through a_{m-1} would be assigned to the other trees in which there is a single subtree connected to the root node. Indices $a_{m-1}+1$ through $a_{m-1} + a_{m-2}$ would be assigned to the trees with exactly two subtrees connected to the root node, such that one of the subtrees has exactly $m-2$ nodes. The next $a_{m-3}a_2 = a_{m-1}$ indices would be assigned to trees with exactly two subtrees connected to the root node, such that one of the subtrees has exactly $m-3$ nodes. See Figure 10 for an example.

To aid the recognition of a watermark we use a special signature could indicates that ``the next thing that follows" is the real watermark. This is similar to the signals used between baseball coaches and their players. See Figure 11 for an example.

One advantageous consequence of the present approach is that obfuscation techniques which target code and static data will have no effect on the dynamic structures that are being built. There are, however, other techniques which can obfuscate dynamic data, and which we will need to tamperproof against. There are three types of obfuscating transformations from which will need to be protected against:

1. An adversary can add extra pointers to the nodes of linked structures. This will make it hard for R to recognize the real graph within a lot of extra bogus pointer fields.

2. An adversary can rename and reorder the fields in the node, again making it hard to recognize the real watermark.

3. Finally, an adversary can add levels of indirection, for example by splitting nodes into several linked parts.

These transformations are illustrated in Figure 12. It is important to note here that obfuscating linked structures has some potentially serious consequences. For example, splitting nodes will increase the dynamic memory requirement of the program (each cell carries a certain amount of overhead for type information etc.), which could mean that a program which ran on, say, a machine with 32M of memory would now not run at all. Furthermore, if we assume that an adversary does not know in which dynamic structure our watermark is hidden, he is going to have to obfuscate every dynamic memory allocation in the entire program.

Next will be discussed techniques for tamperproofing a dynamic watermark against the obfuscation attacks outlined above.

The types of tamperproofing techniques that will be available will depend on the nature of the distributed object code. If the code is strongly typed and supports reflection (as is the case with Java bytecode) we can use these reflection capabilities to construct the tamperproofing code. If, on the other hand, the application is shipped as stripped, untyped, native code (as is the case with most programs written in C, for example) this possibility is not open to us. Instead, we can insert code which manipulates the dynamically allocated structures in such a way that obfuscating them would be unsafe.

ANSI C's address manipulation facilities and limited reflection capabilities allow for some trivial tamperproofing checks:

```
include <stdlib.h>
include <stddef.h>
struct s int a; int b;;
void main ()
     if (offsetof(struct s, a) >
          offsetof(struct s, b)) die();
     if (sizeof(struct s) != 8) die();
```

```
}
```

These tests will cause the program to terminate if the fields of the structure are reordered, or the structure is split or augmented.

Figure 13 (a) shows how Java's reflection package allows us to perform similar tamperproofing checks.Note that this example code is not completely general, since Java does not specify the relative order of class fields.

Figure 13 (b) shows how we can also use opaque predicates and variables to construct code which appears to (but in fact, does not) perform ``unsafe" operations on graph nodes. A de-watermarking tool will not be able to statically determine whether it is safe to apply optimizing or obfuscating transformations on the code. In the example in Figure 13 (b), V is an opaque string variable whose value is "car", although this is difficult for a de-watermarker to work out statically. At 1 it appears as if some (unknown to the de-watermarker) field is being set to null, although this will never happen. The statement 2 is a redundant operation performing n.car = n.car, although (due to the opaque variable R whose value is always 1) this cannot in general be worked out statically.

For increased obscurity, the code to build the watermark should be scattered over the entire application. The only restriction is that when the end of the input sequence $I=I_1 \ldots I_k$ is reached, all individual parts $W_1 \ldots W_{k-1}$ of the watermarking structure have been built and assembled into the complete watermark :

if (input =) W_1 =...;

if (input =) $W_2 = \ldots;$

if (input =) $W_1 = ;\ W_1 \oplus W_1 \oplus \ldots \oplus W_{k-1}$

In order to identify the watermark structure, the recognizer must have access to the runtime stack and heap and be able to enumerate all heap objects. If this is not directly supported by the runtime environment (as, for example, is the case with Java), we have two choices. We can either rewrite the runtime system to give us the necessary functionality or we can provide our own memory allocator. Notice, though, that this is only necessary when we are attempting to recognize the watermark. Under normal circumstances the application can run on the standard runtime system.

A number of techniques are known in the prior art for hiding copyright notices in the object code of a program. It is the inventors' belief that such methods are not resilient to attack by obfuscation -- an adversary can apply a series of transformations that will hide or obscure the watermark to the extent that it can no longer be reliably retrieved.

The present invention indicates that the most reliable place to hide a watermark is within the dynamic (stack and heap) structures of the program, as it is being executed with a particular input sequence.

Where in the foregoing description reference has been made to elements or integers having known equivalents, then such equivalents are included as if they were individually set forth.

Although the invention has been described by way of example and with reference to particular embodiments, it is to be understood that modifications and/or improvements may be made without departing from the scope or spirit of the invention.

Auckland Uniservices Ltd

[illegible]

By its Attorneys

Baldwin Shelston Waters

INTELLECTUAL PROPERTY OFFICE
OF N.Z.

10 JUN 1998

RECEIVED

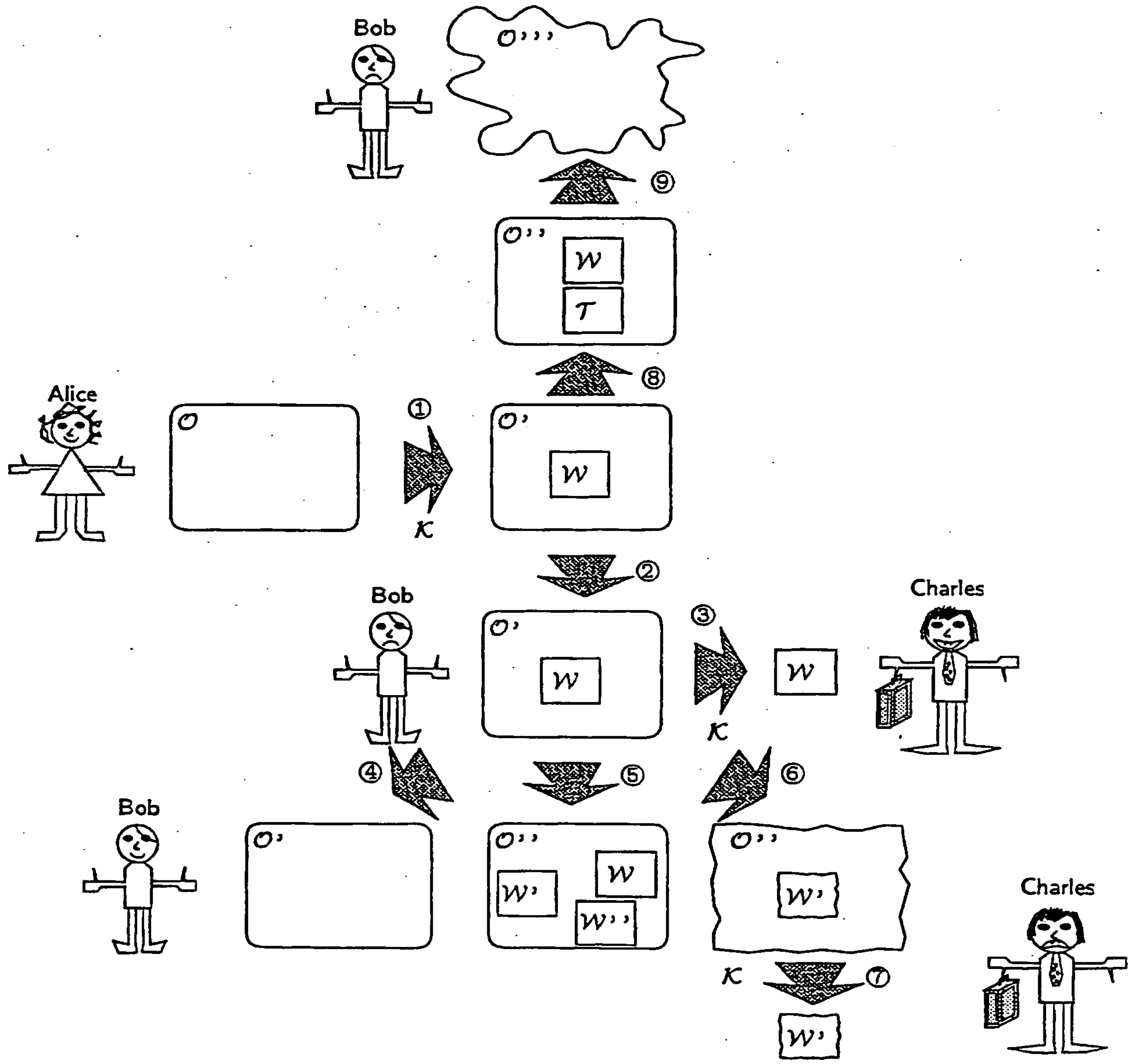
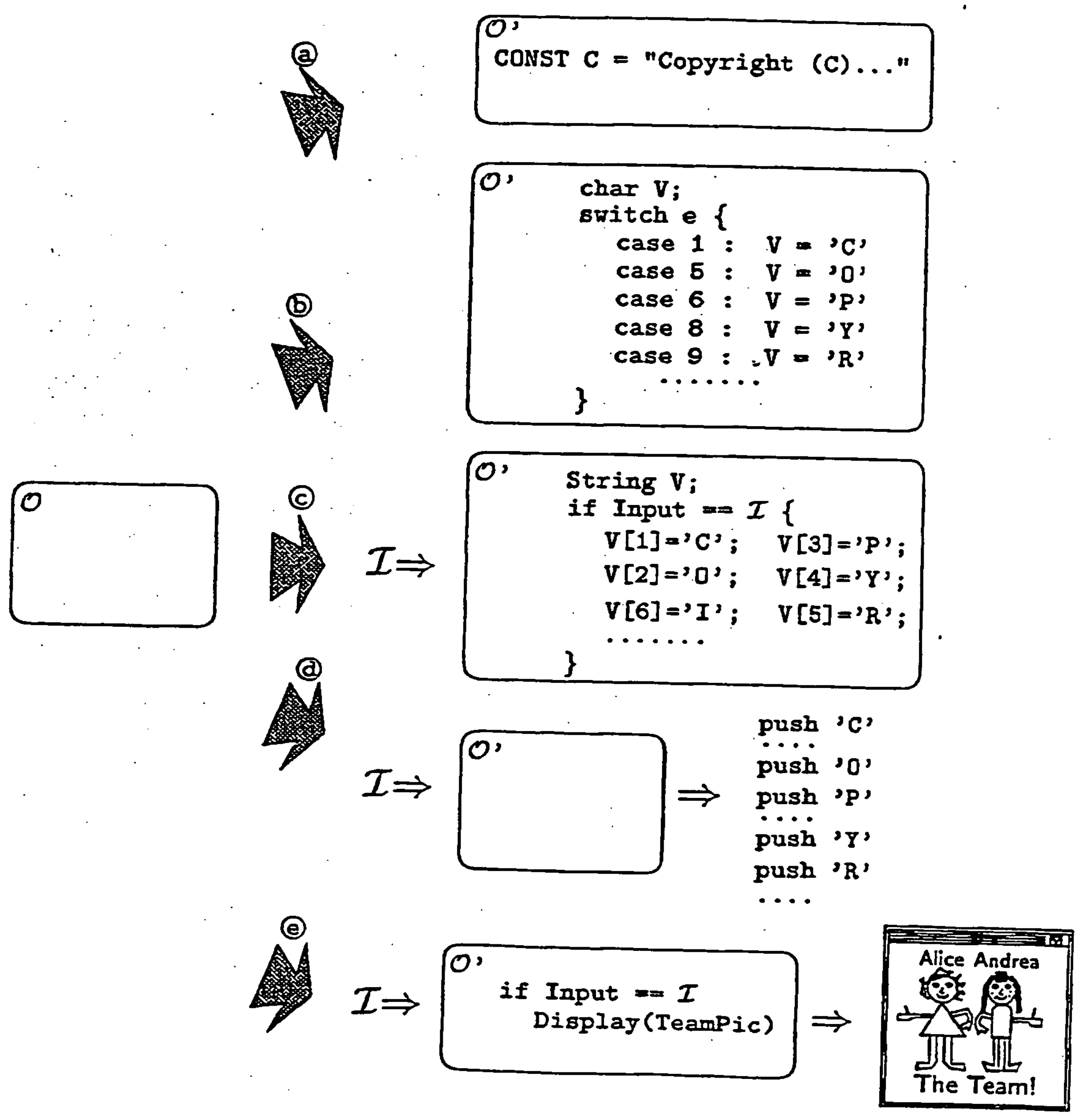

Bob
O'''
⑨
O''
W
T
⑧
Alice
O
①
K
O'
W
②
Bob
O'
W
③
K
W
Charles
④
⑤
⑥
Bob
O'
O''
W'
W
W''
O''
W'
K
⑦
W'
Charles

FIGURE 1



```
CONST C = "Copyright (C)..."
```

```
char V;
switch e {
   case 1 :  V = 'C'
   case 5 :  V = 'O'
   case 6 :  V = 'P'
   case 8 :  V = 'Y'
   case 9 :  V = 'R'
      .......
}
```

```
String V;
if Input == ℐ {
   V[1]='C';   V[3]='P';
   V[2]='O';   V[4]='Y';
   V[6]='I';   V[5]='R';
   .......
}
```

```
push 'C'
....
push 'O'
push 'P'
....
push 'Y'
push 'R'
....
```

```
if Input == ℐ
   Display(TeamPic)
```

FIGURE 2

```
String G (int n) {
   int i=0,k;
   String S;
   while (1) {
      L1:  if (n==1) {S[i++]="A";k=0;goto L6};
      L2:  if (n==2) {S[i++]="B";k=-2;goto L6};
      L3:  if (n==3) {S[i++]="C";goto L9};
      L4:  if (n==4) {S[i++]="X";goto L9};
      L5:  if (n==5) {S[i++]="C";goto L11};
            if (n>12) goto L1;
      L6:  if (k++<=2) {S[i++]="A";goto L6} else goto L8;
      L8:  return S;
      L9:  S[i++]="C"; goto L10;
      L10:  S[i++]="B"; goto L8;
      L11:  S[i++]="C"; goto L12;
      L12:  goto L10;
   }
}
```

FIGURE 3

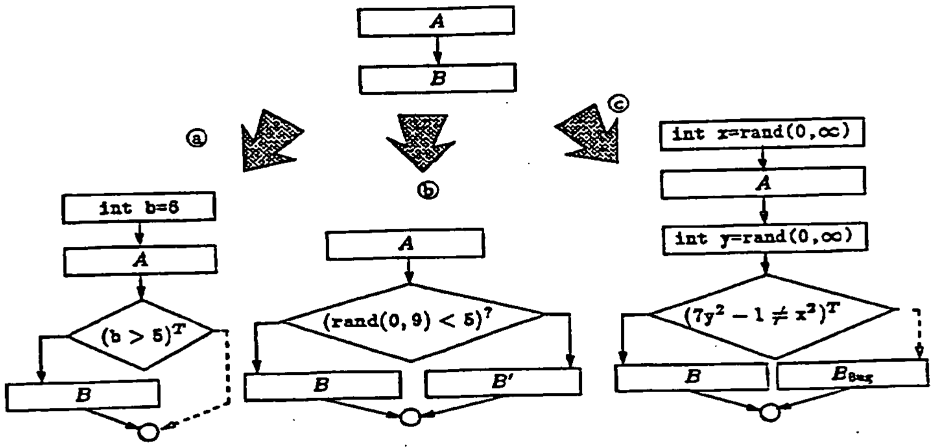


FIGURE 4

$g(V)$ p	$g(V)$ q	$f(p,q)$ V	$2p+q$
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

	AND[A,B]	A 0	A 1	A 2	A 3
	0	3	0	0	0
B	1	3	1	2	3
	2	0	2	1	3
	3	3	0	0	3

```
(1) bool A,B,C;
(2) B = False;
(3) C = False;
(4) C = A & B;
(5) C = A & B;
(6) if (A) ···;
(7) if (B) ···;
```

$$\xRightarrow{\mathcal{T}}$$

```
(1') short a1,a2,b1,b2,c1,c2;
(2') b1=0; b2=0;
(3') c1=1; c2=1;
(4') x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(5') c1=(a1 ^ a2) & (b1 ^ b2); c2=0;
(6') x=2*a1+a2; if ((x==1) || (x==2)) ···;
(7') if (b1 ^ b2) ···;
```

FIGURE 5

$$\begin{aligned}
Z(X+r,Y) &= 2^{32}\cdot Y+(r+X) &&= Z(X,Y)+r\\
Z(X,Y+r) &= 2^{32}\cdot(Y+r)+X &&= Z(X,Y)+r\cdot 2^{32}\\
Z(X\cdot r,Y) &= 2^{32}\cdot Y+X\cdot r &&= Z(X,Y)+(r-1)\cdot X\\
Z(X,Y\cdot r) &= 2^{32}\cdot Y\cdot r+X &&= Z(X,Y)+(r-1)\cdot 2^{32}\cdot Y
\end{aligned}$$

```
(1) int X=45;
    int Y=95;
(2) X += 5;
(3) Y += 11;
(4) X *= c;
(5) Y *= d;
```

$$\xRightarrow{\mathcal{T}}$$

```
(1') long Z=167759086119551045;

(2') Z += 5;
(3') Z += 47244640256;
(4') Z += (c-1)*(Z & 4294967295);
(5') Z += (d-1)*(Z & 18446744069414584320);
```

FIGURE 6

```
int Sum(int A[]) {
  int i, sum=0;
  int n=A.length;
  for (i=0;i<n;i++)
    sum += A[i];
  return sum;
}
```

$\overset{T}{\Longrightarrow}$

```
int Sum(int A[]) {
  int sum=0, i=0, pc=0;
  int s[]=new int[5], sp=-1;
  loop:  while (true)
    switch("fcgabced".charAt(pc)) {
      case 'a':  sum += s[sp--]; pc++; break;
      case 'b':  i++; pc++; break;
      case 'c':  s[++sp] = i; pc++; break;
      case 'd':  if (s[sp--] > s[sp--]) pc -= 6;
                 else break loop; break;
      case 'e':  s[++sp] = A.length; pc++; break;
      case 'f':  pc += 5; break;
      case 'g':  s[sp] = A[s[sp]]; pc++; break;
    }
  return sum;
}
```

FIGURE 7

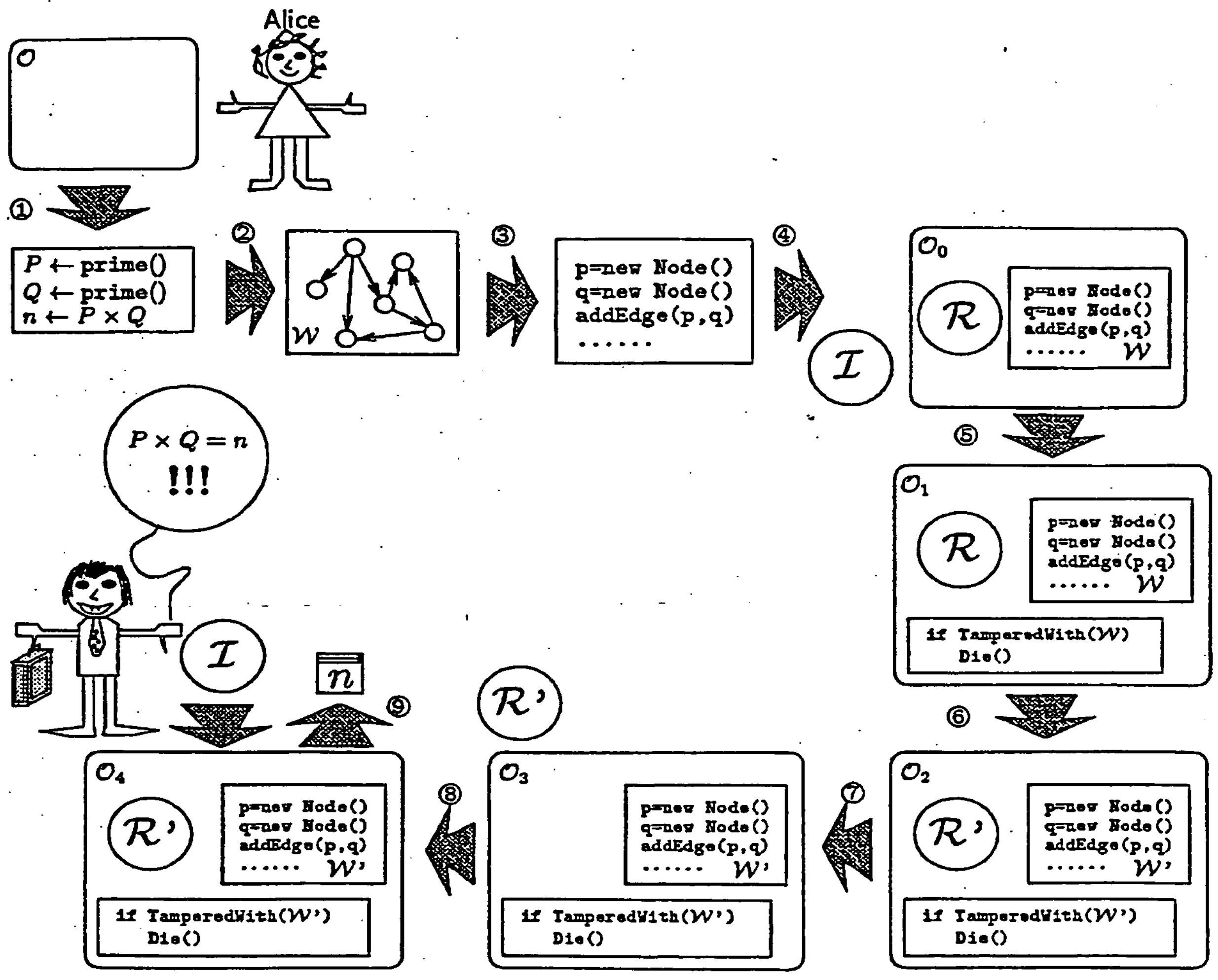
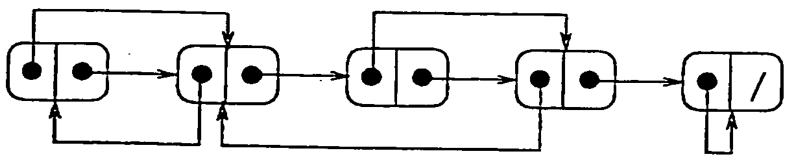

Alice
$\mathcal{O}$
①
$P \leftarrow$ prime()
$Q \leftarrow$ prime()
$n \leftarrow P \times Q$
②
$\mathcal{W}$
③
p=new Node()
q=new Node()
addEdge(p,q)
......
④
$\mathcal{I}$
$\mathcal{O}_0$
$\mathcal{R}$
$\mathcal{W}$
⑤
$\mathcal{O}_1$
if TamperedWith($\mathcal{W}$)
Die()
⑥
$\mathcal{O}_2$
$\mathcal{R}$'
$\mathcal{W}$'
if TamperedWith($\mathcal{W}$')
Die()
⑦
$\mathcal{O}_3$
⑧
$\mathcal{O}_4$
⑨
n
$P \times Q = n$
!!!

FIGURE 8



$$3\cdot 6^4 \quad + \quad 2\cdot 6^3 \quad + \quad 3\cdot 6^2 \quad + \quad 4\cdot 6^1 \quad + \quad 1\cdot 6^0 = 4453 = 61 * 73$$

FIGURE 9

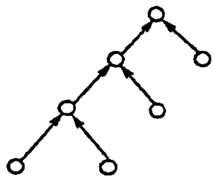


FIGURE 10

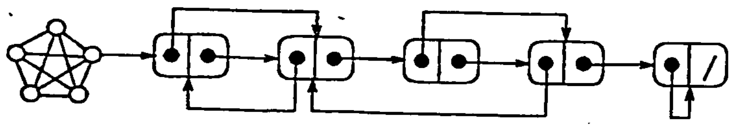


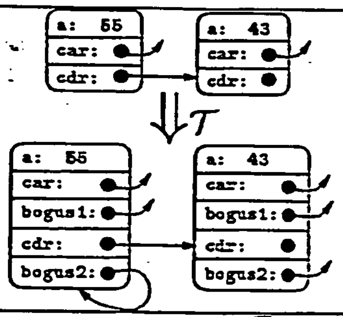
FIGURE 11

①

```
class T {
   int a;
   T car;
   T cdr;
}
```

$\xrightarrow{\mathcal{T}}$

```
class T {
   int a;
   T car;
   T bogus1;
   T cdr;
   T bogus2;
}
```

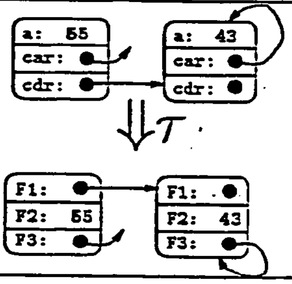


②

```
class T {
   int a;
   T car;
   T cdr;
}
```

$\xrightarrow{\mathcal{T}}$

```
class T {
   T F1;
   int F2;
   T F3;
}
```



③

```
class T {
   int a;
   T car;
   T cdr;
}
      .......
n = new T;
```

$\xrightarrow{\mathcal{T}}$

```
class T {
   int a;
   T1 bogus;
}
class T1 {
   T car;
   T cdr;
}
      ......
n = new T;
n.bogus = new T1;
```

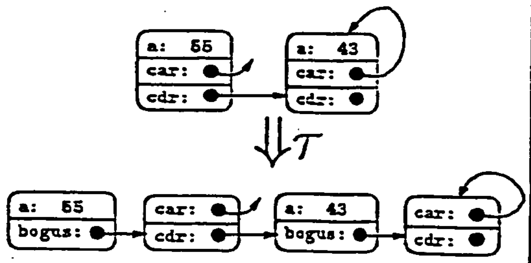


FIGURE 12

```
class C {public int a; public C car, cdr;}

public static void main(String[] args) {
   Field[] F = C.class.getFields();
   if (F.length != 3)
      die();
   if (F[0].getType() !=
      java.lang.Integer.TYPE)
      die();
   if (F[1].getType() != C.class)
      die();
   if (F[2].getType() != C.class)
      die();
}
```

(a)

```
class C {public int a; public C car, cdr;}

public static void main(String[] args)
      throws NoSuchFieldException,
      IllegalAccessException {
   Field f;
   String V;
   C n = new C();
   Class c = n.getClass();
   if (P^F) {
      f = c.getField(V="car");
      ① f.set(n, null);
   }

   Field F = c.getFields();
   int R;
   ② F[R=1].set(n, n.car);
}
```

(b)

FIGURE 13